**Guía Completa de Ejercicios con Promesas en JavaScript**

¡Bienvenido/a! Esta guía está diseñada para llevarte desde los conceptos más básicos hasta un entendimiento sólido de las **Promesas** en JavaScript. Las promesas son la base del manejo de la asincronía en el JavaScript moderno, y dominarlas te abrirá las puertas a la creación de aplicaciones más complejas y robustas.

**1. El Problema: El Código Asíncrono y el "Callback Hell"**

Antes de las promesas, el código asíncrono (código que no se ejecuta inmediatamente, como una llamada a una API o la lectura de un archivo) se manejaba principalmente con **callbacks**. Un callback es una función que se pasa como argumento a otra función, para ser ejecutada más tarde.

Cuando tenías múltiples operaciones asíncronas que dependían una de la otra, terminabas con algo así, conocido como **"Callback Hell"** o "Pirámide de la Muerte":

// Ejemplo de Callback Hell (¡esto es lo que las promesas vienen a solucionar!)  
operacionAsincrona1(function(resultado1) {  
  operacionAsincrona2(resultado1, function(resultado2) {  
    operacionAsincrona3(resultado2, function(resultado3) {  
      operacionAsincrona4(resultado3, function(resultado4) {  
        // Y así sucesivamente...  
        console.log("¡Terminado!");  
      });  
    });  
  });  
});

Este código es difícil de leer, de mantener y de manejar en caso de errores.

**2. ¿Qué es una Promesa?**

Una **Promesa (Promise)** es un objeto que representa la eventual finalización (o fallo) de una operación asíncrona. En lugar de pasar un callback, las funciones asíncronas pueden devolver una promesa.

Imagina que pides una pizza. El restaurante te da un ticket (la **promesa**). No tienes la pizza todavía, pero tienes la promesa de que la recibirás.

Una promesa tiene 3 estados posibles:

1. **Pendiente (pending):** El estado inicial. La operación aún no se ha completado (aún no te han entregado la pizza).
2. **Cumplida (fulfilled o resolved):** La operación se completó con éxito (¡tu pizza ha llegado!). La promesa ahora tiene un **valor** (la pizza).
3. **Rechazada (rejected):** La operación falló (se equivocaron de pedido o hubo un problema). La promesa ahora tiene una **razón** o **error** (el motivo del fallo).

Una vez que una promesa se cumple o se rechaza, su estado nunca más cambiará. Es inmutable.

**3. Consumiendo Promesas: .then(), .catch() y .finally()**

Para trabajar con el resultado de una promesa, usamos sus métodos.

**Mini-Documentación:**

* **.then(onFulfilled, onRejected):** Se ejecuta cuando la promesa se cumple. Recibe una función (onFulfilled) cuyo primer argumento es el valor de resolución de la promesa. Opcionalmente, puede recibir una segunda función (onRejected) para manejar el rechazo, aunque es más común usar .catch() para eso.
* **.catch(onRejected):** Se ejecuta **solo si** la promesa es rechazada. Recibe una función (onRejected) cuyo primer argumento es la razón del rechazo (el error). Es el lugar ideal para manejar errores.
* **.finally(onFinally):** Se ejecuta siempre, sin importar si la promesa fue cumplida o rechazada. No recibe ningún argumento y es útil para tareas de limpieza (ej: ocultar un spinner de carga).

Ejercicio 3.1: Promesa Exitosa

Tarea: A continuación, tienes una función que simula una llamada exitosa a una API y devuelve una promesa. Usa .then() para imprimir el mensaje de éxito en la consola.

const promesaExitosa = new Promise((resolve, reject) => {  
  // Simulamos una operación que toma tiempo  
  setTimeout(() => {  
    resolve("¡Operación exitosa!");  
  }, 2000); // Se resuelve después de 2 segundos  
});  
  
// Tu código aquí para consumir la promesa

Ejercicio 3.2: Promesa Fallida

Tarea: Ahora tienes una promesa que siempre falla. Usa .catch() para capturar el error e imprimirlo en la consola.

const promesaFallida = new Promise((resolve, reject) => {  
  setTimeout(() => {  
    reject("¡Algo salió muy mal!");  
  }, 1500);  
});  
  
// Tu código aquí para consumir la promesa

Ejercicio 3.3: Usando .then y .catch juntos

Tarea: Consume la promesaExitosa del ejercicio 3.1, pero esta vez, añade también un bloque .catch (aunque no se ejecutará, es una buena práctica tenerlo). Luego, consume la promesaFallida del ejercicio 3.2 usando tanto .then como .catch. Observa qué se ejecuta en cada caso.

**Muestra de Solución:**

// console.log("Consumiendo promesa exitosa:");  
// promesaExitosa  
//   .then(resultado => {  
//     console.log("Éxito:", resultado);  
//   })  
//   .catch(error => {  
//     console.log("Este catch no se ejecutará:", error);  
//   });  
  
// console.log("\nConsumiendo promesa fallida:");  
// promesaFallida  
//   .then(resultado => {  
//     console.log("Este then no se ejecutará:", resultado);  
//   })  
//   .catch(error => {  
//     console.error("Error:", error);  
//   });

Ejercicio 3.4: El Bloque .finally()

Tarea: Consume la promesaExitosa y la promesaFallida. En ambos casos, añade un bloque .finally() que imprima el mensaje "La operación ha finalizado." en la consola. Observa que este mensaje aparece en ambos casos.

**4. Creando Nuestras Propias Promesas**

Mini-Documentación:

Para crear una promesa, usamos su constructor: new Promise(executor).

El executor es una función que recibe dos argumentos: resolve y reject. Ambas son funciones.

* Dentro del executor, realizas tu operación asíncrona.
* Si la operación es exitosa, llamas a resolve(valor) para cumplir la promesa con un valor.
* Si la operación falla, llamas a reject(error) para rechazar la promesa con un error.

function simularDescarga(url) {  
  return new Promise((resolve, reject) => {  
    console.log(`Iniciando descarga de ${url}...`);  
    setTimeout(() => {  
      if (url.includes("error")) {  
        reject(`Error al descargar ${url}`);  
      } else {  
        resolve(`Contenido de ${url}`);  
      }  
    }, 2500);  
  });  
}

Ejercicio 4.1: Función que Devuelve una Promesa

Tarea: Crea una función verificarNumero(numero) que devuelva una promesa.

* Si el numero es mayor o igual a 10, la promesa debe cumplirse (resolve) con el mensaje "El número es válido.".
* Si el numero es menor que 10, la promesa debe rechazarse (reject) con el mensaje "El número no es válido.".  
  Prueba tu función con un número válido (ej: 15) y uno no válido (ej: 5), consumiendo las promesas con .then y .catch.

Ejercicio 4.2: Promesa con Retardo Aleatorio

Tarea: Crea una función esperaAleatoria() que devuelva una promesa. La promesa debe resolverse después de un tiempo aleatorio entre 1 y 5 segundos. El valor de resolución debe ser el número de milisegundos que esperó.

**Muestra de Solución:**

// function esperaAleatoria() {  
//   return new Promise((resolve, reject) => {  
//     const tiempoEspera = Math.floor(Math.random() \* 4000) + 1000; // Entre 1000 y 5000 ms  
//     setTimeout(() => {  
//       resolve(tiempoEspera);  
//     }, tiempoEspera);  
//   });  
// }  
  
// console.log("Esperando un tiempo aleatorio...");  
// esperaAleatoria().then(tiempo => {  
//   console.log(`La promesa se resolvió después de ${tiempo} milisegundos.`);  
// });

**5. Encadenamiento de Promesas (Chaining)**

La verdadera magia de las promesas aparece cuando las encadenas. Un bloque .then() puede devolver:

* Un valor: Este valor se pasará como argumento al siguiente .then().
* Otra promesa: El siguiente .then() esperará a que esta nueva promesa se resuelva, y recibirá su valor de resolución.

Esto nos permite crear secuencias de pasos asíncronos de forma legible.

// Ejemplo de encadenamiento  
Promise.resolve(10)  
  .then(numero => {  
    console.log("Paso 1:", numero); // 10  
    return numero \* 2; // Devuelve un valor  
  })  
  .then(numero => {  
    console.log("Paso 2:", numero); // 20  
    // Devuelve una nueva promesa  
    return new Promise(resolve => setTimeout(() => resolve(numero \* 2), 1000));  
  })  
  .then(numero => {  
    console.log("Paso 3:", numero); // 40 (después de 1 segundo)  
  })  
  .catch(err => console.error(err));

Ejercicio 5.1: Proceso en Varios Pasos

Tarea: Simula un proceso en tres pasos:

1. Crea una función paso1() que devuelva una promesa que se resuelve después de 1 segundo con el valor 1.
2. Encadena un .then() que reciba ese valor, lo imprima, y devuelva la promesa de una función paso2(valor) que se resuelve después de 1 segundo con valor + 1.
3. Encadena otro .then() que reciba el nuevo valor, lo imprima, y devuelva la promesa de una función paso3(valor) que se resuelve después de 1 segundo con valor + 1.
4. Finalmente, imprime el resultado final.

Ejercicio 5.2: Encadenamiento con Manejo de Errores

Tarea: Crea una cadena de promesas donde uno de los pasos puede fallar.

1. Empieza con Promise.resolve(10).
2. El primer .then multiplica el número por 2 y lo devuelve.
3. El segundo .then verifica si el número es mayor que 30. Si lo es, rechaza la promesa con "Número demasiado grande". Si no, devuelve el número.
4. El tercer .then imprime "Proceso exitoso con el número: [numero]".
5. Añade un .catch al final para capturar cualquier error en la cadena.  
   Prueba cambiando el valor inicial para que la promesa se cumpla y se rechace.

**6. async / await: Una Forma Más Limpia de Usar Promesas**

async/await es "azúcar sintáctico" sobre las promesas. No reemplaza a las promesas, sino que te permite escribir código asíncrono que parece síncrono, lo cual es mucho más fácil de leer.

**Mini-Documentación:**

* **async:** La palabra clave async se pone antes de una declaración de función para convertirla en una función asíncrona. Una función async **siempre** devuelve una promesa.
* **await:** La palabra clave await **sólo se puede usar dentro de una función async**. Hace que la ejecución de la función async se pause hasta que la Promise a su derecha se resuelva, y luego reanuda la ejecución con el valor de resolución.
* **Manejo de Errores:** Se usa el bloque try...catch estándar de JavaScript para manejar rechazos.

// Función que devuelve una promesa  
function obtenerUsuario(id) {  
  return new Promise(resolve => {  
    setTimeout(() => {  
      resolve({ id: id, nombre: "Juan" });  
    }, 1000);  
  });  
}  
  
// Función async que usa await  
async function mostrarUsuario(id) {  
  console.log("Buscando usuario...");  
  try {  
    const usuario = await obtenerUsuario(id);  
    console.log(`Usuario encontrado: ${usuario.nombre} (ID: ${usuario.id})`);  
  } catch (error) {  
    console.error("Error al buscar usuario:", error);  
  }  
  console.log("Fin de la función async.");  
}  
  
// mostrarUsuario(123);

Ejercicio 6.1: Convertir .then a async/await

Tarea: Toma la solución del ejercicio 5.1 (Proceso en Varios Pasos) y reescríbela dentro de una función async usando await.

Ejercicio 6.2: Múltiples awaits

Tarea: Escribe una función async llamada obtenerDatosCompletos().

1. Debe llamar (await) a una función obtenerUsuario() que devuelve una promesa que se resuelve con un objeto de usuario después de 1 segundo.
2. Debe llamar (await) a una función obtenerPosts(usuario.id) que devuelve una promesa que se resuelve con un array de posts después de 1.5 segundos.
3. Finalmente, debe imprimir el nombre del usuario y la cantidad de posts que tiene.

**Muestra de Solución:**

// function obtenerUsuario() {  
//   return new Promise(resolve => {  
//     setTimeout(() => resolve({ id: 1, nombre: "Leanne Graham" }), 1000);  
//   });  
// }  
  
// function obtenerPosts(userId) {  
//   return new Promise(resolve => {  
//     setTimeout(() => resolve([  
//       { userId: userId, title: "Post 1" },  
//       { userId: userId, title: "Post 2" }  
//     ]), 1500);  
//   });  
// }  
  
// async function obtenerDatosCompletos() {  
//   console.log("Iniciando obtención de datos...");  
//   try {  
//     const usuario = await obtenerUsuario();  
//     console.log(`Usuario obtenido: ${usuario.nombre}`);  
//     const posts = await obtenerPosts(usuario.id);  
//     console.log(`Posts obtenidos: ${posts.length}`);  
//     console.log(`${usuario.nombre} tiene ${posts.length} posts.`);  
//   } catch (error) {  
//     console.error("Error obteniendo datos completos:", error);  
//   }  
// }  
  
// obtenerDatosCompletos();

Ejercicio 6.3: Manejo de Errores con async/await

Tarea: Crea una función async intentarOperacionRiesgosa(). Dentro de un bloque try, llama (await) a una función operacionQueFalla() que siempre devuelve una promesa rechazada. En el bloque catch, imprime el error capturado.

**7. Métodos Estáticos de Promise**

Estos métodos funcionan sobre un iterable (generalmente un array) de promesas.

**Mini-Documentación:**

* **Promise.all(iterable):** Espera a que **todas** las promesas del iterable se cumplan. Si **una** falla, toda la promesa Promise.all se rechaza inmediatamente con el error de esa promesa.
* **Promise.race(iterable):** Se resuelve o rechaza tan pronto como **la primera** de las promesas del iterable se resuelve o se rechaza.
* **Promise.allSettled(iterable):** Espera a que **todas** las promesas se establezcan (se cumplan o se rechacen). Devuelve un array de objetos que describen el resultado de cada promesa.
* **Promise.any(iterable):** Se cumple tan pronto como **la primera** de las promesas del iterable se cumple. Se rechaza solo si **todas** las promesas del iterable son rechazadas.

Ejercicio 7.1: Promise.all

Tarea: Simula la descarga de tres archivos al mismo tiempo. Crea un array con tres promesas, cada una con un setTimeout de diferente duración. Usa Promise.all para esperar a que todas terminen y luego imprime un mensaje de "Todas las descargas completadas" junto con los resultados.

**Muestra de Solución:**

// const descarga1 = new Promise(res => setTimeout(() => res("Archivo 1"), 1000));  
// const descarga2 = new Promise(res => setTimeout(() => res("Archivo 2"), 2000));  
// const descarga3 = new Promise(res => setTimeout(() => res("Archivo 3"), 1500));  
  
// const promesas = [descarga1, descarga2, descarga3];  
  
// console.log("Iniciando todas las descargas...");  
// Promise.all(promesas)  
//   .then(resultados => {  
//     console.log("Todas las descargas completadas.");  
//     console.log("Resultados:", resultados); // ["Archivo 1", "Archivo 2", "Archivo 3"]  
//   })  
//   .catch(err => console.error("Una de las descargas falló:", err));

Ejercicio 7.2: Promise.race

Tarea: Simula una carrera entre dos servidores para ver cuál responde primero. Crea dos promesas, una que se resuelve en 1 segundo y otra en 2 segundos. Usa Promise.race para obtener el resultado de la más rápida.

Ejercicio 7.3: Promise.allSettled

Tarea: Tienes un array de promesas donde algunas se cumplen y otras se rechazan. Usa Promise.allSettled para procesarlas todas y luego itera sobre los resultados para imprimir el estado y el valor/razón de cada una.

Ejercicio 7.4: Promise.any

Tarea: Simula la búsqueda de un recurso en tres servidores de respaldo. El primero falla, el segundo tiene éxito y el tercero también. Usa Promise.any para obtener el resultado del primer servidor que responda con éxito.

Has cubierto todos los aspectos fundamentales de las promesas en JavaScript. La práctica constante de estos conceptos te convertirá en un desarrollador mucho más capaz y preparado para enfrentar los desafíos del código asíncrono.